Extending the logical update view with transaction support

Jan Wielemaker

Web and Media group, VU University Amsterdam,
De Boelelaan 1081a,
1081 HV Amsterdam, The Netherlands,
J.Wielemaker@vu.nl

Abstract. Since the database update view was standardised in the Prolog ISO standard, the so called logical update view is available in all actively maintained Prolog systems. While this update view provided a well defined update semantics and allows for efficient handling of dynamic code, it does not help in maintaining consistency of the dynamic database. With the introduction of multiple threads and deployment of Prolog in continuously running server applications, consistency of the dynamic database becomes important.
In this article, we propose an extension to the generation-based implementation of the logical update view that supports transactions. Generation-based transactions have been implemented according to this description in the SWI-Prolog RDF store. The aim of this paper is to motivate transactions, outline an implementation and generate discussion on the desirable semantics and interface prior to implementation.

1 Introduction

Although Prolog can be considered a deductive database system, its practice with regard to database update semantics is rather poor. Old systems typically implemented the *immediate update view*, where changes to the clause-set become immediately visible to all goals on backtracking. This update view implies that a call to a dynamic predicate must leave a choice point to anticipate the possibility that a clause is added that matches the current goal. Quintus introduced[1] the notion of the *logical update view* [4], where the set of visible clauses for a goal is frozen at the start of the goal, which allows for pruning the choice-point on a dynamic predicate if no more clauses match the current goal. Through the ISO standard (section 7.5.4 of ISO/IEC 1321 l-l), the logical update view is adopted by all actively maintained implementations of the Prolog language.

The logical update view helps to realise efficient programs that depend on the dynamic database, but does not guarantee consistency of the database if multiple changes of the database are required to realise a transition from one consistent state to the next. A typical example is an application that realises a transfer between two accounts as shown below.

[1] http://dtai.cs.kuleuven.be/projects/ALP/newsletter/archive_93_96/net/systems/update.htm

```
transfer(From, To, Amount) :-
        retract(balance(From, FromBalanceStart)),
        retract(balance(To, ToBalanceStart)),
        FromBalance is FromBalanceStart - Amount,
        ToBalance is ToBalanceStart + Amount,
        asserta(balance(From, FromBalance)),
        asserta(balance(To, ToBalance)).
```

Even in a single threaded environment, this code may be subject to timeouts, (resource-)exceptions and exceptions due to programming errors that result in an inconsistent database. Many Prolog programs contain a predicate to clean the dynamic database to avoid the need to restart the program during development. Still, such a predicate needs to be kept consistent with the used set of dynamic predicates and provides no easy solution if the database is not empty at the start.

Obviously, in concurrent applications we need additional measures to guarantee consistency with concurrent transfer requests and enquiries for the current balance. Below we give a possible solution based on mutexes. Another solution is to introduce a `bank` thread and realise transfer as well as enquiries with message passing to the `bank` thread.

```
mt_transfer(From, To, Amount) :-
        with_mutex(bank, transfer(From, To, Amount)).

mt_balance(Account, Balance) :-
        with_mutex(bank, balance(Account, Balance)).
```

As we need to serialise update operations in a multi threaded context, update operations that require significant time to complete seriously harm concurrency.

With transactions, we can rewrite the above using the code below, where we do not need any precautions for reading the current balance.[2] Consistency is guaranteed because, as we will explain later, two transactions that retract the same clause are considered to conflict.

```
mt_transfer(From, To, Amount) :-
        transaction(transfer(From, To, Amount), true, [restart(true)]).
```

In the remainder of this article, we first describe related work. Next, we define the desired semantics of transactions in Prolog, followed by a description how these semantics can be realised using generations. In section 5, we propose a concrete set of predicates to make transactions available to the Prolog programmer. We conclude with implementation experience in the SWI-Prolog RDF store and a discussion section.

[2] Reading multiple values from a single consistent view still requires a transaction. See section 3.1.

2 Related work

The most comprehensive overview of transactions in relation to Prolog we found is [2], which introduces "Transaction logic". Transaction logic has been implemented as a prototype for XSB Prolog [3]. This is a much more fundamental solution in dealing with update semantics than what we propose in this article. In [2], we also find descriptions of related work, notably *Dynamic Prolog* and an extension to Datalog by Naqvi and Krishnamurthy. These systems too introduce additional logic and are not targeted to deal with concurrency.

Contrary to these systems, we propose something that is easy to implement on engines that already provide the logical update view and is easy to understand and use for a typical Prolog programmer. What we do learn from these systems is that a backtrackable dynamic database has promising applications. We not propose to support backtracking modifications to the dynamic database yet, but our proposal simplifies later implementation thereof, while the transaction interface may provide an adequate way to scope backtracking. See **transaction/3** described in section 5.

3 Transaction semantics

Commonly seen properties of transactions are known by the term ACID,[3] summarised below. We want to realise all these properties, except for *Durability*.

Atomic Either all modifications in the transaction persist or none of the modifications.

Consistency A successful transaction brings the system from one consistent state into the next.

Isolation The modifications made inside a transaction are not visible to the outside world before the transaction is committed. Concurrent access always sees a consistent database.

Durability The effects of a committed transaction remain permanently visible.

In addition, code that is executed in the context of a transaction should behave according to the traditional Prolog (logical view) update semantics and it must be possible to nest transactions, such that code that creates a transaction can be called from any context, both outside and inside a transaction.

An obvious baseline interface for dealing with transactions is to introduce a meta-predicate `transaction`(*:Goal*). If *Goal* succeeds, the transaction is committed. If *Goal* fails or throws an exception, the transaction is rolled back and **transaction/1** fails or re-throws the exception. In our view, **transaction/1** is logically equivalent to **once/1** (i.e., it prunes possibly remaining choice points) because database actions are still considered side-effects. Too much real-world code is intended to be deterministic, but leaves unwanted choice points. This is also the reason why **with_mutex/2** prunes choice points.

Note that it is easy to see useful application scenarios for non-deterministic transactions. For example, generate-and-test applications that use the database could be implemented using the skeleton code below. To support this style of programming, failing

[3] `http://en.wikipedia.org/wiki/ACID`

into a transaction should atomically make the changes invisible to the outside and all changes after the last choice point inside the transaction must be discarded. This can be implemented by extending each choice point with a reference into the change-log maintained by the current transaction.

```
generate_and_test :-
        transaction(generate_world),
        satisfying_world,
        !.
```

Given our proposed once-based semantics, we can still improve considerably on this use-case compared to traditional Prolog using a side-effect free generator. This results in the following skeleton:

```
generate_and_test :-
        generate_world(World),
        transaction(( assert_world(World),
                      satisfying_world
                    )),
        !.
```

3.1 Snapshots

We can exploit the isolation feature of transactions to realise *snapshots*. A predicate `snapshot`(*:Goal*) executes *Goal* as **once/1** without globally visible affects on the dynamic database. This feature is a supplement to SWI-Prolog's *thread local* predicates, predicates that have a different set of clauses in each thread. Snapshots provide a comfortable primitive for computations that make temporary use of the dynamic database. At the same time they make such code thread-safe as well as safe for failed or incomplete cleanup due to exceptions or programming errors.

Snapshots also form a natural abstraction to read multiple values from a consistent state of the dynamic database. For example, the summed balance of a list of accounts can be computed using the code below. The snapshot isolation guarantees that the result correctly represents that summed balance at the time that the snapshot was started.

```
summed_balance(Accounts, Sum) :-
        snapshot(maplist(balance, Accounts, Balances)),
        sum_list(Balances, Sum).
```

Note that this sum may be outdated before the isolated goal finishes. Still, it represents a figure that was true at a particular point in time, while unprotected execution can compute a value that was never correct. For example, consider the sequence of events below, where to summed balance is 10$ too high.

1. The 'summer' fetches the balance of A
2. A concurrent operation transfers 10$ from A to B
3. The 'summer' fetches the balance of B

4 Generation based transactions

A common technique used to implement the Prolog logical update view is to tag each clause with two integers: the generation in which it was born and the generation in which it died. A new goal saves the current generation and only considers clauses created before and not died before its generation. This is clearly described in [1]. Below, we outline the steps to add transactions to this picture.

First, we split the generation range into two areas: the low values, 0..G_TBASE (transaction base generation) are used for globally visible clauses. Generations above G_TBASE are used for generations, where we split the space further by thread-id. E.g., the generation for the 10th modification inside a transaction executed by thread 3 is G_TBASE+3*G_TMAX+10.

Isolation Isolated behaviour inside a transaction is achieved by setting the modification generation to the next thread write generation. Code operating outside a transaction does not see these modifications because they are time-stamped 'in the future'. Code operating inside the transaction combines the global view at the start of the transaction with changes made inside the transaction, i.e., changes in the range G_TBASE+⟨*tid*⟩*G_TMAX..G_TBASE+(⟨*tid*⟩+1)*G_TMAX.

Atomic Committing a transaction renumbers all modifications to the current global write generation and then increments the generation. This implies that commit operations must be serialised (locked). All modifications become atomically visible at the moment that the global generation is incremented. If a transaction is discarded, all asserted clauses are made available for garbage collection and the generation of all retracted clauses is reset to infinity.

Consistency The above does not provide consistency guarantees. We add a global consistency check by disallowing multiple retracts of the same clause. This implies that an attempt to retract an already retracted clause inside a transaction or while the transaction commits causes the transaction to be aborted. This constraint ensures that code that *updates* the database by retracting a value, computing the new value and asserting this becomes safe. For example, this deals efficiently with global counters or the balance example from section 1. Note that disallowing multiple retracts is also needed because there is only one placeholder to store the 'died generation'. Similar to relational databases, we can add an integrity constraint, introducing `transaction`(*:Goal, :Constraint*), where *Constraint* is executed while the global commit lock is held.

Nesting Where we need distinct generation ranges for concurrently executing transactions, we can use the generation range of the parent transaction for a nested transaction because execution as **once/1** guarantees strict nesting. Nested transactions merely need to remember where the nested transaction started. Committing is a no-op, while discarding is the same as discarding an outer transaction, but only affecting modifications after the start of the nested transaction.

Implication for visibility rules The logic to decide that a clause is visible does not change for queries outside transactions because manipulations inside transactions are 'in the future'. Inside a transaction, we must exclude globally visible clauses that have died inside the transaction (i.e., between the transaction start generation and the current generation) and include clauses that are created and not yet retracted in the transaction.

5 Proposed predicates

We propose to add the following three predicates to Prolog. In the description below, predicate arguments are prefixed with a *mode annotation*. The `:` annotation means that the argument is module-sensitive, e.g., *:Goal* means that *Goal* is called in the module that calls the transaction interface predicate. The modes `+`, `-` and `?` specifies that the argument is 'input', 'output' and 'either input or output'.

transaction(*:Goal***)**
: Execute *Goal* in a transaction. *Goal* is executed as by **once/1**. Changes to the dynamic database become visible atomically when *Goal* succeeds. Changes are discarded if *Goal* does not succeed.

transaction(*:Goal, :Constraint***)**
: Run *Goal* as **transaction/1**. If *Goal* succeeds, execute *Constraint* while holding the `transaction` mutex (see **with_mutex/2**[4]). If *Constraint* succeeds, the transaction is committed. Otherwise, the transaction is discarded. If *Constraint* fails, throw the error `error(transaction_error(constraint, failed), _)`. If *Constraint* throws an exception, rethrow this exception.

transaction(*:Goal, :Constraint, +Options***)**
: As **transaction/3**, processing the following options:

  restart(*+Boolean***)**
  : If `true`, catch errors that unify with `error(transaction_error(_,_), _)` and restart the transaction.

  id(*Term***)**
  : Give the transaction an identifier. This identifier is made available through **transaction_property/2**. There are no restrictions on the type or instantiation of *Term*.

snapshot(*:Goal***)**
: Execute *Goal* as **once/1**, isolating changes to the dynamic database and discarding these changes when *Goal* completes, regardless how.

transaction_property(*?Transaction, ?Property***)**
: True when this goal is executing inside a transaction identified by the opaque ground term *Transaction* and has given *Property*. Defined properties are:

  level(*-Level***)**
  : Transaction is nested at this level. The outermost transaction has level 1. This property is always present.

  modified(*-Boolean***)**
  : True if the transaction has modified the dynamic database.

[4] `http://www.swi-prolog.org/pldoc/doc_for?object=with_mutex/2`

modifications(-*List*)
: List expresses all modifications executed inside the transaction. Each element is either a term `retract`(*Term*),`asserta`(*Term*) or `assertz`(*Term*). Clauses that are both asserted and erased inside the transaction are omitted.

id(?*Id*)
: Transaction has been given the current *Id* using **transaction/3**.

If any of these predicate encounters a conflicting retract operation, the exception `error(transaction_error(conflict, PredicateIndicator), _)` is generated.

6 Implementation results: the SWI-Prolog RDF-DB

The SWI-Prolog RDF database [5] is a dedicated C-based implementation of a single dynamic predicate `rdf`(*?Subject, ?Predicate, ?Object*). The dedicated implementation was introduced to reduce memory usage and improve performance by exploiting known features of this predicate. For example, all arguments are ground, and *Subject* and *Predicate* are know to be atoms. Also, all 'clauses' are unit clauses (i.e., there are no rules). Quite early in the development of the RDF store we added transactions to provide better consistency and grouping of modifications. These features were crucial for robustness and 'undo' support in the graphical triple editor Triple20 [6]. Initially, the RDF store did not support concurrency. Later, this was added based on 'read/write locks', i.e., multiple readers or one writer may access the database at any point in time.

With version 3[5] of the RDF store, developed last year, we realised the logical update view also for the external **rdf/3** predicate and we implemented transactions following this article. In addition, we realised concurrent garbage collection of dead triples. The garbage collector examines the running queries and transactions to find the oldest active generation and walks the linked lists of the index hash-tables, removing dead triples from these lists. Actual reclaiming of the dead triples is left to the Boehm-Demers-Weiser conservative garbage collector.[6]

7 Discussion

We have described an extension to the generation-based logical update view available in todays Prolog system that realises transactions. There is no additional memory usage needed for clauses. The engine (each engine in multi threaded Prolog systems) is required to maintain a stack of transaction records, where each transaction remembers the global generation in which is was created and set of affected clauses (either asserted or retracted). The visibility test of a clause for goals outside transactions is equal to the test required for realising the logical update view and requires an additional test of the same cost if a transaction is in progress.

[5] Available from `http://www.swi-prolog.org/git/packages/semweb.git`, branch `version3`.

[6] `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

The described implementation of transactions realises ACI of the ACID model (atomic, consistency and isolation, but not durability). Durability can be realised by using a constraint goal that uses **transaction_property/2** to examine the modifications and write the modifications to a journal file or external persistent store.

Our implementation has two limitations: (1) goals in a transaction are executed as **once/1** (pruning choice-points) and (2) it is not possible for multiple transactions to retract the same clause. Supporting non-deterministic transactions requires additional changes to Prolog choice points, but transactions already maintain a list of modifications to realise commit and rollback and **transaction/3** already provides an extensible interface to activate this behaviour. Supporting multiple retracts is possible by using a list to represent the 'died' generations of the clause. This is hardly useful for transactions because multiple concurrent retracts indicate a conflicting update. However, this limitation is a serious restriction for snapshots (section 3.1).

We have implemented this transaction system for the SWI-Prolog RDF store, where it functions as expected. We believe that transactions will greatly simplify the implementation of concurrent programs that use the dynamic database as a shared store. At the same time it eliminates the need for serialisation of code, improving concurrent performance. In our experience with Triple20, transactions are useful in single threaded applications to maintain consistency of the database. Notably, consistency of the dynamic storage is maintained when an edit operation fails due to a programming error or an abort initiated from the debugger. This allows for fixing the problem and retrying the operation without restarting the application.

We plan to implement the outlined features in SWI-Prolog in the near future.

Acknowledgements

This research was partly performed in the context of the COMBINE project supported by the ONR Global NICOP grant N62909-11-1-7060. This publication was supported by the Dutch national program COMMIT.

I would like to thank Jacco van Ossenbruggen, Michiel Hildebrand and Willem van Hage for their feedback in redesigning the transaction support for the SWI-Prolog RDF store.

References

1. Egon Boerger and Bart Demoen. A framework to specify database update views for prolog. In Jan Maluszynski and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 147–158. Springer Berlin / Heidelberg, 1991. 10.1007/3-540-54444-5_95.
2. Anthony J. Bonner, Michael Kifer, and Mariano Consens. Database programming in transaction logic. In *In Proc. 4th Int. Workshop on Database Programming Languages*, pages 309–337, 1993.
3. Samuel Y.K. Hung. Implementation and performance of transaction logic in prolog. Master's thesis, Department of Computer Science, University of Toronto, 1996.
4. Timothy G. Lindholm and Richard A. O'Keefe. Efficient implementation of a defensible semantics for dynamic prolog code. In *ICLP*, pages 21–39, 1987.

5. Jan Wielemaker, Guus Schreiber, and Bob J. Wielinga. Prolog-based infrastructure for rdf: Scalability and performance. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 644–658. Springer, 2003.
6. Jan Wielemaker, Guus Schreiber, and Bob J. Wielinga. Using triples for implementation: The triple20 ontology-manipulation tool. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 773–785. Springer, 2005.